1-1-2006

Utilizing timing error detection and recovery to dynamically improve superscalar processor performance

Mikel Anton Bezdek
Iowa State University

Utilizing timing error detection and recovery to dynamically improve superscalar processor performance

by

Mikel Anton Bezdek

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Arun Somani, Major Professor
Manimaran Govindarasu
Yan-bin Jia

Iowa State University

Ames, Iowa

2006

Graduate College
Iowa State University

This is to certify that the master's thesis of

Mikel Anton Bezdek

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

To provide reliable execution, traditional design methodologies perform timing error avoidance. Worst case parameters are assumed when determining a processor's operating frequency, allowing the maximum propagation delay through the system to be met. However, in practice the worst cases are rare, leading to a large amount of exploitable performance improvement if timing errors can be detected and recovered from. To this end, we propose a novel low cost scheme which allows a superscalar processor to dynamically tune its frequency past the worst case limit. When timing errors occur, they are detected and recovered from locally. Additionally, the number of errors that occur are monitored by one of several sampling methods. When the error rate becomes too high, leading to decreased performance, the frequency is scaled back. Experimental results show an average performance gain of 45% across all benchmark applications. The cost of implementing the error detection and recovery is kept modest by reusing the existing pipeline logic to detect the timing errors.

CHAPTER 1. INTRODUCTION

To provide reliable execution, traditional design methodologies for synchronous circuits perform timing error avoidance. Clock periods are selected to give enough time for the worst case operation of a circuit. However, worst case conditions are rare, leading to a large amount of possible performance improvement. In order to take advantage of this improvement and maintain reliable execution, the circuit must be made in some way tolerant to timing errors.

1.1 Background

A representation of a typical circuit is shown in figure 1.1. Operation begins when a clock edge at register A stores the incoming data, sending it to the combinational logic. The logic has until the next clock edge to compute the result, at which time it will be stored in register B. A timing error occurs when the clock edge occurs before the input of B has stabilized to the correct value. In this case, an incorrect value will be stored in B and sent to the output, and, depending on the importance of the data from B, may cause a failure. To prevent this from occurring, designers have traditionally focused on avoiding timing errors through careful selection of the clock period.

When selecting the clock period for a circuit such as in figure 1.1, the longest path through the logic from register A to register B is found. The maximum propagation delay of this path is calculated and is combined with the delay through A as well as with the setup time of B. This number gives a lower bound on the achievable clock period. By fixing the clock period longer than this delay, the designer can be reasonably sure that timing errors will be avoided. The designer is only reasonably sure, however, because there are many variables that affect the observed propagation delay through the circuit. In fact, to achieve an acceptable level of

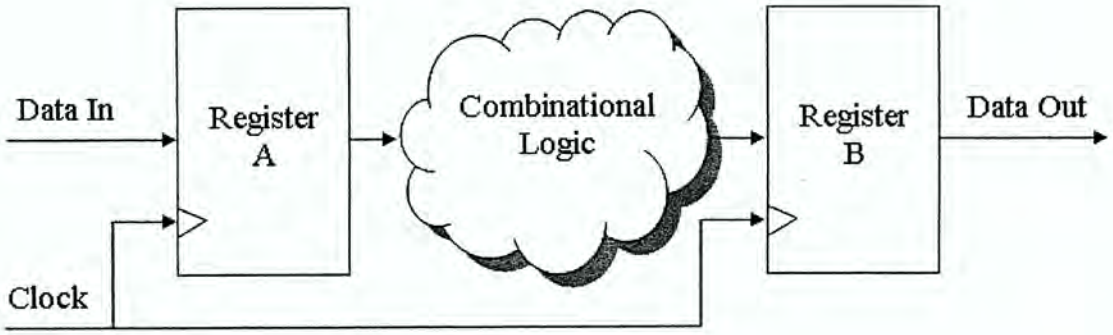

Figure 1.1 Typical Synchronous Circuit

confidence with traditional timing error avoidance, the worst case parameters of these variables must be assumed.

The variables affecting propagation delay can be divided into process variations, present when a circuit is fabricated, and environmental variations, present when a circuit is in operation. At fabrication, imperfect equipment and variability in process parameters, such as the doping concentration, lead to both inter–die and intra–die variability. Variations occur in the delay characteristics of the same circuit fabricated on different dies (inter–die variability) and in the delay through different gates of a circuit on a single die (intra–die variability). To account for these variations, designers often assume delays three sigma from the typical delay, Nassif (2001). Additionally, while in operation, environmental conditions such as temperature and power supply voltage have an effect on the delay through any path in a circuit. These conditions can only be estimated when fixing the clock period of a circuit, so again, designers must opt for the worst case to ensure that most of the time, the circuit will execute free of timing errors.

The result of accounting for these variations is the selection of a clock period that is overly conservative. Rarely will the worst case combination of all influencing factors occur together. Another important consideration is that even when operating in the worst possible conditions, the worst case delay through the circuit will only be observed if the longest path is sensitized. Consider the case of a simple ripple–carry adder. For this circuit, the longest path is a carry

generated at the first bit position and propagated through all remaining bits. However, as observed by Austin et al. (2005), a carry–chain of this sort is very rare for both random and application generated input vectors. The vast majority of the observed carry–chains were under seven bits long. Finally, even if the longest circuit delay path is sensitized, it will, in fact, only produce an error if the input vector causes the output to change from its previous value. If the output of the circuit is latched before the bit position with the longest path settles, there is still a 50 percent chance that the random data latched was correct. Of course, this probability decreases exponentially as the number of bit positions that may be in error increases. Given these factors, the typical clock period achievable by circuits is much lower, and thus the performance much better than the worst case bound.

When the circuit under consideration is a modern pipelined processor, the worst case methodology has special implications. A pipelined processor divides the complex circuit needed to execute program instructions into stages, allowing better utilization of the hardware and increasing the throughput of the processor. In this situation, the clock period of the entire circuit is limited by the worst case delay through the longest stage. This means that if one stage is significantly longer than the rest, the performance of the processor as a whole will suffer. Techniques currently employed to remedy this situation, such as super–pipelining, which divides longer stages further into sub–stages, have their limitations as will be seen in chapter 2.

An entire community has sprung up centered around pushing the performance of modern processors past the levels set by worst case design. This phenomenon is known commonly as overclocking, Colwell (2004), and comprises at–home users manually accelerating the frequency of their computers past the limit set at production. Impressive results have been achieved; Overclockers.com (2006) reports that an AMD processor with a worst case frequency set at 2.2 GHz has been successful overclocked at speeds of up to 3.1 GHz. However, this impressive speed boost comes at a cost. Overclockers know that they have gone too far only when the system crashes. So, while overclocking does demonstrate that significant room for performance improvement exists, it is not a viable solution for the vast majority of computer users who

expect their system to behave in a reliable manner.

A somewhat more conservative approach than overclocking seeks to exploit the performance gap left by worst case design parameters, while at the same time providing reliable execution. This approach, coined "better than worst–case design" Austin et al. (2005), uses principles from fault tolerance, employing some combination of spatial and temporal redundancy. This redundancy, in turn, allows timing errors to be caught and recovered from, and is thus able to maintain reliable execution while operating at a point past the worst case limits. In some sense, these methodologies, described in detail in chapter 2, perform overclocking with a safety net.

1.2 Contribution

This thesis presents a solution which addresses the limitations imposed by worst case design called SPRIT^{3}E, or Superscalar PeRformance Improvement Through Tolerating Timing Errors. This framework allows the clock frequency of a superscalar processor to be dynamically tuned to its optimal value, beyond the worst case limit. Because the frequency is dynamically modified as the processor is running, variations in the environmental conditions, such as temperature and voltage, as well as variations present from fabrication, are automatically adjusted for. At some point, as frequency scales faster and faster, timing errors will begin to occur. To prevent these errors from corrupting the execution of the processor, fault tolerance in the form of temporal redundancy is proposed. Specifically, pipeline stages are augmented with a dual latching scheme.

To mitigate the timing errors that occur at higher frequencies, every pipeline stage is augmented with a second, time–delayed register. The clock for this register is phase shifted such that the combinational logic is effectively given its full, worst case propagation delay time to execute. The output from both the main and the delayed registers are compared at every cycle to detect when a timing error has occurred. When the values differ, the error recovery mechanism takes over, supplying the correct value from the delayed register. Additionally, this value must be forwarded to any other instructions dependent on instruction in which the

timing error occurred. Because errors are detected quickly and the recovery technique utilizes many existing paths through the processor, the area and performance overhead incurred from allowing timing errors to occur is kept to a minimum.

To evaluate this technique, several experiments were performed. First, to explore the possibilities of dynamic frequency scaling, an 18x18 multiplier was operated at varying frequencies, and the number of resultant timing errors was observed. Next, having developed a dual latching framework and showing an achievable frequency 44% faster than the worst case level with the multiplier, the technique was applied to a superscalar processor. Using a superscalar processor synthesized in an FPGA, the frequency and application dependent timing error behavior was analyzed. Then with these results, the ability of the SPRIT^{3}E methodology to provide performance improvement was determined for varying error sampling implementations. For long term execution, all benchmark applications showed an achievable performance improvement of 45%.

1.3 Organization

The rest of this thesis is organized as follows: Chapter 2 provides a detailed review of the literature relating to the topic of pipeline performance and better than worst case design. The error mitigation technique proposed, including the detection and recovery from timing errors as well as the applications of this scheme to other types of errors, is described in chapter 3. In chapter 4, a description of the clocking system used to generate the dynamically modifiable clock is given. The experimental framework and results are presented in chapter 5, and chapter 6 concludes the thesis.

CHAPTER 2. LITERATURE REVIEW

Previous relevant work can be divided into three groups. The first comprises efforts that seek to improve pipelined processor performance while staying within the worst case design limitations. The second group consists of techniques that use redundancy in some form or another to detect, and in some cases correct, errors. The final and most relevant group is the "better than worst–case design" strategies, which utilize fault tolerance in some manner to push performance past the limits set by worst case estimates.

2.1 Designs to Improve Pipeline Performance

Performance has long been one of the main factors driving new designs. Because the traditional design methodology has assumed a clock frequency fixed at the worst case propagation delay, many techniques have been developed to improve performance without violating this assumption. Three important and well–known techniques are device scaling, superpipelining, and asynchronous circuits.

By far, the most common method to improve processor performance is that of device scaling. In this method, the transistors used to implement the circuit are made smaller, allowing not only more transistors to be placed on a chip, but also allowing each transistor to switch at a faster rate. This technique is one of the main contributors to the fantastic performance gains made by modern integrated circuits, as devices have scaled from 10μm to below 100nm in just a few decades, Allan et al. (2002). However, as observed by Yeap (2002) and Frank et al. (2001), the limits of device scaling are rapidly approaching. Limitations imposed by dynamic leakage currents and fabrication methods will soon set a bound on the performance gains achievable by device scaling alone.

Another method used to improve the performance of a processor within the worst case design parameters is superpipelining. First appearing in the MIPS R4000 processor, Mirapuri et al. (1992), superpipelining allows a faster clock frequency by dividing the clock limiting stages into multiple sub–stages. In the MIPS R4000, memory accesses, previously given only one cycle to execute, were divided into two stages. By doing this, each new memory stage was made slightly faster than the ALU stage, allowing an increase in overall frequency. This technique, also referred to as deeper pipelining, has been vigorously applied to modern processors. For example, progressing from the original five stages of the first pipelined designs, the Intel P4 processor now utilizes twenty pipeline stages, Hinton et al. (2001). However, there are limits to the benefits of superpipelining as well. With deeper and deeper pipelines, the penalties for mis–predicted branches become ever higher. This is due both to an increased branch resolution latency, i.e. branches must pass through more stages before the mis–prediction is detected, and to an increased latency to refill the pipeline following the necessary flush incurred by a incorrect prediction. Additionally, adding more stages increases the penalties imposed by other hazards present in the pipeline. To find a balance between these inverse performance factors, it has recently been shown by Hartstein and Puzak (2002) that while the optimal number of pipeline stages is somewhat application dependent, on average performance is maximized with around twenty stages. So, although superpipelining has worked well in the past, here too there is a fundamental limit to performance gains.

In contrast to the above techniques, asynchronous designs sidestep the clock frequency issue all together. In an asynchronous design, a stage may begin execution as soon as the data is ready from the previous stage. Emerson (1997) presents a good overview of asynchronous design methodology. In the past, these methodologies have suffered from a lack of tools supporting CAD for asynchronous systems. Additionally, since mainstream thinking has focused on synchronous designs, asynchronous methodology has often been viewed as hard to comprehend, especially since it requires a great deal of attention be placed on the dynamic state of the circuit, Hauck (1993). However, in recent years promising developments have been made as asynchronous designs have begun to gain acknowledgment. Amde et al. (2003) presents a

tool automated design of an asynchronous DLX processor. It seems that, although still in their early stages of acceptance, asynchronous designs offer a promising alternative to the worst case limits that plague traditional synchronous designs.

2.2 Fault Tolerant Designs

In addition to performance, reliable execution has also been a main factor influencing the design process. To provide reliability, a multitude of methods has been developed. All of these methods use redundancy in some form to determine when and where errors have occurred. Fault tolerance can be implemented in both software and hardware, though for the purpose of this thesis, only the hardware methods will be presented. The types of hardware redundancy can be broadly classified into information redundancy, spatial redundancy, and temporal redundancy.

In a fault tolerant design utilizing information redundancy, extra information, in the form of bits, is added to allow detection of errors in the original data. To prevent having to duplicate the entire data, encoding techniques such as parity, cyclic redundancy check (CRC), or Hamming codes are used. The codes differ in the number of redundant bits that must be added, as well as in their ability to detect and correct errors. This type of fault tolerance has been widely applied to provide protection for memory data, as well as to provide error checking when transmitting data.

With spatial redundancy, tolerance to faults is provided by duplicating some or all of the actual circuitry used to compute a result. To detect the occurrence of an error, only one duplicate copy is necessary. Both copies are given the inputs, and if the outputs ever disagree, then an error has occurred. However, to decide which copy has made the error, at least one more copy of the circuit is needed. This observation leads directly to possibly the most common inception of spatial redundancy, triple modular redundancy (TMR), Von Neumann (1966). As implied, TMR uses three copies of a circuit to both detect and correct errors. The outputs of the copied circuits are sent to a voter, which, in turn, picks the most common answer. In this system, any error in any one of the circuits will be masked as long as the other

two execute correctly. To provide increased tolerance to faults, extensions to TMR have been proposed, such as replication of the voting circuitry, as well as using more than three copies of the circuit, or N–modular redundancy. Spatial redundancy provides rapid recovery from errors, minimizing performance overhead, but does so at the cost of a high area overhead.

Alternately, temporal redundancy provides fault tolerance by using the same circuitry to re–compute results at a later time. Similar to spatial redundancy, the multiple results are compared to determine an error. However, while spatial methods can tolerate both transient and permanent errors, temporal techniques will only work for transients. This is because a permanent error in the circuit produces a consistent faulty result, which is indistinguishable in the multiple copies created through temporal redundancy. For this reason, temporal schemes developed for processors are designed to provide tolerance to transient sources of errors, such as single event upsets caused by high energy particle strikes.

A proposed application of temporal redundancy is called REdundant Execution using Spare Elements, or REESE, Nickel and Somani (2001). In this application, all instructions reaching the commit stage in a superscalar processor are re–executed by the same hardware in a redundant stream. This application makes use of idle capacity in the processor to detect transient errors, but incurs a high performance cost for programs with high hardware utilization. To offset the performance costs, spare elements may be added to the processor to prevent the execution of the redundant stream from interfering with the primary instructions.

2.3 Better Than Worst–Case Designs

Noticing the limitations of performance improvement methods that assume worst case operation, a new area of research seeks to use fault tolerance to allow synchronous circuits to perform at typical, "better than worst–case" levels.

A good summary of the principles of this new methodology is presented in Austin et al. (2005). The work presents the RAZOR and DIVA architectures, discussed in detail below, as examples of "better than worst–case" design. Additionally, the authors perform a detailed analysis of a 64 bit adder. They find that due to the carry–chain patterns of both random and

application generated data, an adder optimized for the typical carry–chain will outperform an adder designed to minimize the worst case delay. Finally, they identify the need for and propose new tools to evaluate these designs. However, aside from reviewing the existing architectures, this paper does not present any new solutions that apply this type of methodology to a processor.

An early work that seeks to increase clock frequency through the application of fault tolerance can be found in Uht (2000). The author presents three methods of increasing performance with added redundancy. The first method triplicates the circuit in order to perform twice as fast. To do this one of the copies runs at the fast clock, while the other two operate error–free at the original frequency on alternating data. The results are compared to detect any timing errors. While feasible, this method has an unacceptably high area overhead. To address this, the second scheme, applied to a pipelined processor, calls for only two duplicate copies of the pipeline to achieve a speedup of two. The speedup is achieved because, although each pipeline runs at the original slow frequency, the program instructions are executed alternately on each pipeline. While it achieves a lower overhead than the first, the area penalty imposed by this scheme is still unacceptably high. The final technique proposes dividing long circuits into sub–stages, similar to superpipelining, and clocking each stage at a phase shift of the original clock. This method will produce a fast result, however, since the errors cannot be checked until the next edge of the slow clock, the effective throughput of the circuit will be equivalent to the original slow frequency. The methods presented here are a good initial attempt in exploiting performance by moving past worst case limits, however, the designs presented are not feasible.

Kim and Somani (2001) present a Selective Series Duplex architecture, or SSD, which applies spatial redundancy to a modern high performance processor. This architecture provides the fault tolerant capabilities of a complete duplex system, in which every instruction is executed on both a main and a redundant processor. In addition, by placing the two processors in series, this scheme is able to leverage the capabilities of the main processor to simplify the design of the redundant processor and thus reduce the total area overhead. For example, the redundant processor need not include advanced branch prediction hardware as only the in-

structions committed by the main processor will be re–executed. Through comparison of both executions, SSD is able to detect errors occurring in either processor. However, because both processors run at the same clock speed, this architecture is not able to tolerate timing errors. If the clock frequency is set past the worst case limit, both executions of an instruction will result in an error.

Similar to SSD, DIVA, proposed by Weaver and Austin (2001), also redundantly executes instructions using a second, simpler processor. However, unlike SSD, the "checker" processor operates at a slower rate than the main processor. This method detects timing errors in the main processor as well as other errors. To recover from a detected error, the DIVA architecture uses the correct value from the checker and must restart the processor at the next PC value. This error detection and correction allows a lessening of the stringent requirements for validation of the main processor. The checker processor is able to benefit from the cache fetches of the main processor. However, care must be taken when designing the system to ensure that the checker does not become a bottleneck. Also, the area overhead of the redundant processor, while less than a complete duplication of the main processor, is still significant. A second, highly related work, Weaver et al. (2002), uses the DIVA architecture to build a processor which dynamically tunes its frequency past the worst case limit, allowing some timing errors to occur and be corrected. The execution of the system was simulated, although performance improvements are not presented. Also, the details of the frequency tuning system are not presented or investigated.

In the superscalar domain, Liu and Lu (2000) consider a technique to overcome the shortfalls of superpipelining using fault tolerance. Three complex logic blocks of a superscalar pipeline are identified as possible clock frequency limiting stages: the alu logic, the register rename logic, and the issue logic. Instead of dividing these stages into sub–stages as per superpipelining, this paper proposes the creation of approximate versions of each limiting logic block. The approximate versions are designed to execute in half the time, and are carefully engineered to produce correct results most of the time. In doing this, the benefits of superpipelining are achieved without the drawbacks of increased penalties. To detect errors, the alu

and register rename logic each require two copies of the original, slow logic. The copies operate on alternating data, similar to the first technique proposed by Uht (2000). The issue logic does not require error checking since it will never produce a wrong result, only fail execute optimally. The recovery method flags incorrect instructions, as well as any instruction dependent on the faulty instruction, until the correct value it rewritten. This recovery works well for the rename stage, where errors are caught before corrupting other instructions. Since the alu stage is closer to the end of the pipeline, however, the recovery method does not work as well because the errors cannot be corrected before dependent instructions begin execution. Still, as long as the accuracy of the approximate logic remains relatively high, simulated performance gains over superpipelining are observed. The main drawback of this technique is the large area overhead needed to detect and correct errors.

Ernst et al. (2003) presents RAZOR, a design that detects and recovers from timing errors in an in–order pipeline, performing dynamic voltage scaling (DVS) past the worst case allowable limit. The technique uses a dual latching scheme similar to that proposed in this thesis, in which critical stages are protected with a second flip–flop using a phase–shifted clock. However, because they are tuning the voltage and not the frequency, the amount of phase shift between the clocks can remain a constant and still the worst case constraints of the circuit will be met in the second latch. Errors are detected by comparing the two flip–flop outputs. Recovery consists of supplying the correct value from the second latch and allowing that instruction to complete, while at the same time flushing all other instructions from the pipeline. A processor augmented with the RAZOR scheme has been fabricated, Das et al. (2005), and shows significant power savings over traditional DVS. This technique has a very modest area overhead, but does incur a performance penalty from flushing the pipeline to perform error recovery.

Finally, Uht (2005) presents a means to dynamically tune the clock frequency of a simple, in–order processor. Redundancy is added to perform timing error avoidance in the form a delay chain that has a worst case propagation delay slightly higher than the worst case delay of the processor. As the processor is in operation, this delay chain is constantly monitored. As long as it operates correctly, the frequency is scaled up. As soon as errors in the delay chain show up,

the frequency is brought back to a stable level, avoiding errors in the actual pipeline circuitry. This scheme was implemented in an FPGA and was tested at varying temperatures. The circuit showed significant performance improvement over worst case estimates and responded well to variations in temperature. The benefit of error avoidance such as this is that the performance overhead associated with error recovery is avoided. However, the drawback of this methodology is that only process and temperature variations are accounted for by the delay chain. In fact, since the worst case path through the logic will rarely be sensitized, the highest frequency reached by this design will still be overly conservative. The best performance could be seen by dynamically tuning the frequency while not avoiding, but detecting and correcting timing errors.

CHAPTER 3. ERROR MITIGATION

To allow a superscalar processor to operate at frequencies past the worst case limit, SPRIT^{3}E uses *dual latching*, adding redundant registers between pipeline stages. Through careful control of the clocks, timing errors can be prevented from affecting the redundant register. By comparing the output of both registers, errors can be detected and proper recovery steps taken to ensure correct operation of the pipeline. Also, though designed for timing error tolerance, SPRIT^{3}E can provide some coverage against single event upsets as well. This chapter presents the specifics of the dual latching methodology.

3.1 Dual Latching

At the heart of the fault tolerance provide by SPRIT^{3}E is the dual latching circuit. This circuit is presented in figure 3.1. In this diagram, *Main clock* is the clock controlling synchronous operation of the pipeline, as would be present in an un–augmented pipeline. *PS clock*, or Phase Shifted clock, has the same frequency, but is phase shifted so that its rising edge occurs shortly after the *Main clock*. A detailed description of the generation, as well as a timing analysis of these clocks can be found in chapter 4. Operation of the dual latching circuit begins when the data from the pipeline logic, *Data In*, is stored in the main register at the rising edge of *Main clock*. At this point, *Data Out* provides the stored value to the next stage, which begins to compute a result. A short time later, the rising edge of *PS clock* will cause the input to be stored in the backup register. The output of the main and backup registers are compared, and if they are equal, then no error has occurred.

A timing error occurs when *Data In* has not settled to its correct value by the rising edge of the *Main clock*, storing an incorrect result in the main register. This scenario is illustrated

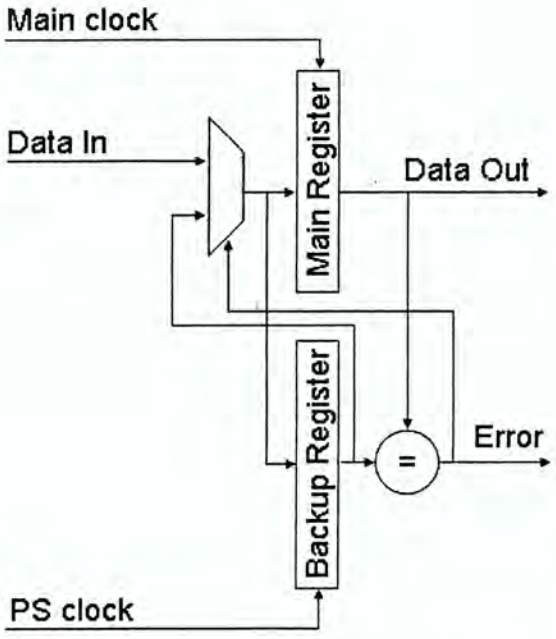

Figure 3.1 Diagram of a Dual Latch

in figure 3.2. The first value received from the logic, *I1*, stabilizes before cycle 1 and continues with no errors. However, the second value, *I2*, does not stabilize until after the rising edge of cycle 2. This causes an unknown value, "xxx" in the figure, to be stored in the main register. By the rising edge of the *PS Clock* in cycle 2, the value has stabilized, and *I2* is stored correctly in the backup register. This stabilization can be guaranteed by setting the phase shift of the *PS clock* to give the logic its full propagation delay. In cycle 2, the mismatch between the main and backup registers will cause an error to be detected at the falling edge of the *PS Clock*. Raising the error signal causes the multiplexer in figure 3.1 to send the data from the backup register as the input to the main register. By doing this, both registers will contain the correct value *I2* by cycle 3. Additionally, the error signal is sent to the global recovery logic, which is responsible for ensuring that wrong value sent in cycle 2 does not lead to a failure.

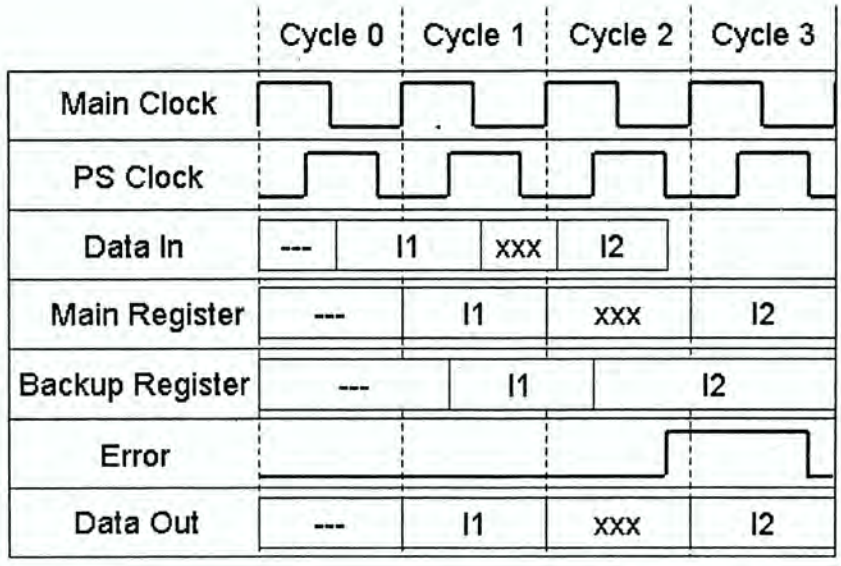

Figure 3.2 Timing Diagram of Dual Latch Error Detection

3.2 Pipeline Error Recovery

In addition to local recovery that takes place in the dual latch via the multiplexer, action must be taken on a global scale as well to maintain correct execution of the pipeline in the event of a timing error. The action taken is dependent on the location of the error. A diagram of the SPRIT^{3}E technique applied to a superscalar processor is shown in figure 3.3. This figure is identical to a superscalar processor whose flip–flops have been replaced by dual latches with one exception. A delay register is added between the output of the re–order buffer and the commit stage. This register prevents an erroneous value from being committed during the clock cycle needed for error detection. One extra stage is added to the pipeline, incurring the minor penalties described for superpipelining, but it is necessary to ensure that the architectural state of the processor remains correct.

Four error locations can be seen in figure 3.3, denoted as *IF error*, *ID error*, *FU_n error*, and *RB error*. Note that as there are multiple function units, there will be multiple error signals from their corresponding dual latches, hence the subscript n. However, in terms of the behavior of the error handler, these errors can be handled in a similar manner. In fact, although the

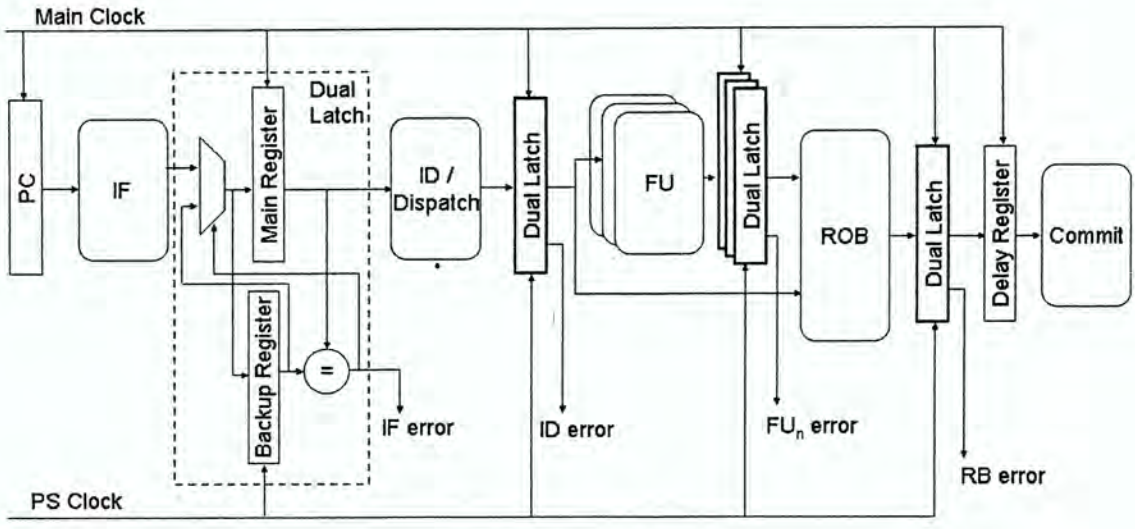

Figure 3.3 A Superscalar Processor Augmented with SPRIT^{3}E

exact steps taken vary by location, there are similar steps that must be followed in every case. Generally, when an error occurs, the incorrect result sent in the previous cycle needs to be cleared. Additionally, the stages preceding the error producing stage need to be notified to prevent data being lost during the cycle in which the error is handled. In the producing stage, the error handling is taken care of by the dual latch hardware as described above. At a high level, this process is very similar to handling a memory hazard in a five stage pipeline, in which stages preceding the load are stalled for a cycle, and a bubble is inserted following the load.

When an error occurs in the instruction fetch stage, the instruction that was sent to the decode stage should not be executed. To prevent execution, the instruction is reduced to a no–op in the decode stage. Additionally, the program counter must be stalled for a cycle, so that following correction of the error, the next instruction will be fetched from the correct address. Finally, since the program counter cannot be updated, any branch or jump instructions attempting to write to PC during the stall cycle must also be stalled for a cycle. All other instructions in the pipeline may be allowed to continue execution.

In the instruction decode and dispatch stage, an error will be propagated to both the re–order buffer, or ROB, as well as to the allocated functional unit. In the ROB, the most recent entry may be cleared by updating the pointer to the head of the buffer. When the next

instruction is dispatched, it will overwrite the faulty instruction. To clear the functional unit, the global error handler must maintain a record of the functional unit used by the dispatcher in the previous cycle. When an error in the dispatch stage is detected, that unit must be cleared to prevent it from writing a wrong value to the re–order buffer. Finally, the signal notifying the instruction fetch stage of a successful dispatch is lowered to prevent the IF stage from fetching the next instruction during the error correction cycle.

An error in the execution of a functional unit will store an incorrect value in the ROB. Additionally, the incorrect value may be forwarded to other functional units whose operands depend on the result of the faulty FU. In the ROB, the instruction must be invalidated to prevent it from being committed. The functional units that have begun execution using the erroneous value must also be stopped. This can be accomplished by sending an error signal using the existing forwarding paths. Finally, the available signal of the faulty functional unit must be lowered to prevent the next instruction from being dispatched to that FU.

An error in the re–order buffer output is prevented from committing in the next cycle by the addition of the delay register mentioned previously. When an error is detected, the delay register is flushed to prevent a faulty commit. Also, the ROB must be prevented from attempting to commit a new instruction in the next cycle. This can be accomplished by manipulating the ready to commit signal from the commit unit.

Figure 3.4 shows the timing of this error recovery when an error in the ALU functional unit occurs. A series of ALU operations is considered, as this is the worst sequence for an error occurring in the ALU. If a different type of instruction was fetched following the ALU instruction causing a timing error, this instruction would be successfully dispatched to a different FU. In the figure, the add instruction completes error free and returns to the ROB. The sub instruction, however, does not stabilize before captured by the main register in cycle 3. This is detected, and the ID stage is prevented from dispatching the or instruction, effectively stalling for one cycle. Additionally, the incorrect value sent to the re–order buffer in cycle 3 will be cleared.

The system shown in figure 3.3 is simplistic in that it assumes only one clock cycle for

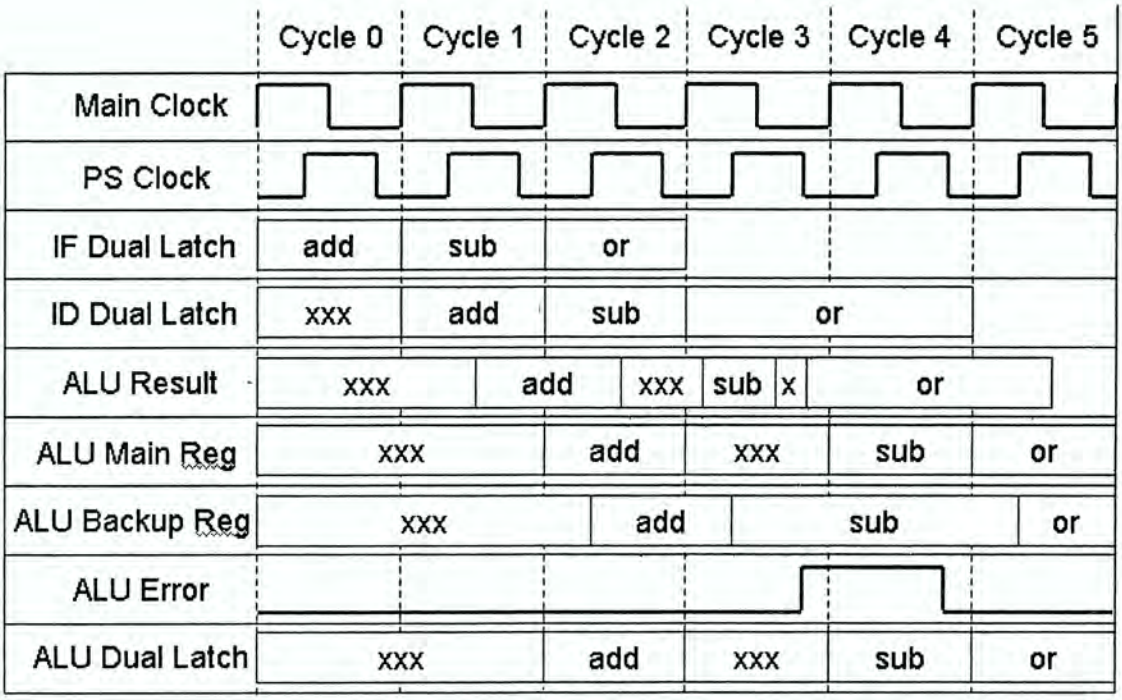

Figure 3.4 A Timing Diagram Showing Recovery from an Error in the ALU

each pipeline stage. However, as was discussed in chapter 2, modern, higher–level pipeline stages are subdivided to smaller, lower–level stages to allow faster clock speeds. The method of error recovery presented here is easily extensible to the superpipelined case. The signals manipulated to stall the high–level stage preceding the error stage will be exercised when the error occurs in any of its low–level stages. Also, if the error occurs in the last low–level stage of a high–level stage, then the methods described above from clearing can be used. Additionally, extra stalling and clearing abilities will need to be added to the low–level stages.

Another important consideration with this design is initialization of the pipeline. When the processor begins execution, the main and backup registers at each stage may contain differing values. This should not be detected as an error until the first rising edge of both the main and PS clocks have reached the registers. Additionally, following a pipeline flush caused by branch mis–prediction, error detection at stage must be stalled until meaningful data again reaches the stage. The delay before beginning error detection will vary between stages, and must be accounted for in the design.

3.3 Single Event Upset Tolerance

The SPRIT^{3}E framework was designed to tolerate timing errors, but the redundancy used can also be applied to tolerate soft errors in the form of single event upsets (SEUs). An SEU is caused when a high energy particle, either from cosmic radiation or decaying radioactive material, strikes the silicon on which the circuit is implemented. If enough charge is deposited by the strike, it will cause the state of a transistor to flip. In memory elements, this directly causes an error to be stored. In combinational logic, the SEU results in a short, 100–200 picosecond wide pulse. If this pulse propagates to the registers at the right time, an error will be stored. Traditionally, memory elements have been more susceptible to SEUs due to their relatively large on–chip area and the natural masking effects of combinational logic. There are three effects that mask SEUs in combinational logic. First, for a pulse in logic to propagate to the registers, it must not be attenuated by the gates it passes through, called *electrical masking*. Also, the pulse must not reach a gate in which it cannot affect the output, called *logical masking*. An example of logical masking is a pulse reaching the input of an AND gate whose other input is driven to zero, forcing the output of the gate to zero regardless of the pulse. Finally, *latching window masking* occurs because the pulse must be present at the input to the register during the clock edge in order for it to be stored. If the pulse attenuates before the clock edge triggers the register, no error will be captured. Although these affects have been sufficient to protect combinational logic for most applications in the past, current trends are reducing their effect and increasing SEU susceptibility, Shivakumar et al. (2002).

To understand how the coverage provided by SPRIT^{3}E differs for SEUs as opposed to timing errors, it is necessary to consider the differences between the two types of soft errors. When compared to SEUs, timing errors are much more predictable. Although the exact amount of time needed to compute a result by a circuit can only be estimated, timing errors will not occur as long as the amount of time provided is equal to or longer that this amount. This is the principle traditional schemes use to avoid timing errors, and the principle that SPRIT^{3}E uses to guarantee correctness of the backup register. SEUs, on the other hand, are uniformly distributed in time and space, meaning that they may occur at any time during a clock cycle.

With respect to SPRIT^{3}E, this means that SEUs may affect either the main register or backup register. By design, SPRIT^{3}E can tolerate any error in the main register, but an error in the backup register will lead to a failure. From the memory perspective, this means that an SEU switching the state of the main register will be detected and corrected. A SEU in the backup register will also cause an error to be detected, however SPRIT^{3}E will assume that the error is in the main register, and thus the SEU will propagate, possibly causing a failure. For SEUs generated in combinational logic, SPRIT^{3}E will extend latching window masking by masking the errors that get stored in the main register. This behavior is illustrated in figure 3.5. As with all synchronous circuits, a portion of the clock period is protected from SEUs by logical masking. Additionally, SEUs occurring between the setup and hold times of the main register, represented by the vertical dashed lines on either side of the rising edge of the main clock, will be detected and corrected by the SPRITE logic. However, a portion of the period remains vulnerable, shown by the solid line.

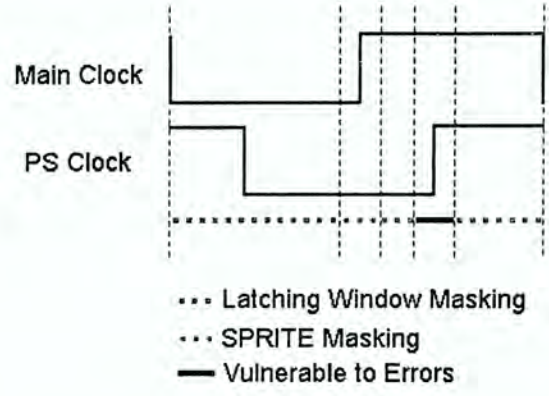

Figure 3.5 SEU Masking In One Period of the Fast Clock

The SPRIT^{3}E technique can be extended to provide greater coverage against SEUs as well as timing errors. To do this, a third register could be added to the dual latch, creating a triple latch, in a manner similar to that presented by Mavis and Eaton (2002). The clock for this register would need to be phase shifted from the PS clock. In addition to the constraints placed on the phase shifts to avoid timing errors, the amount of each phase shift needs to be more than the maximum length of a SEU generated pulse. By doing this, an SEU occurring at any

point in the clock cycle will affect at most one of the registers. The output of all three registers can then be sent to a voter. In the case of a timing error, both backup registers will contain the correct value, and in the case of an SEU, only one of the three registers will be affected. This augmentation will provide greater error coverage, but necessitates more complex clock generation and error detection.

3.4 Performance and Coverage Analysis

Due to the predictability of timing errors, SPRIT^{3}E is able to achieve high coverage and fast recovery at a modest area cost. By allowing the full worst case propagation delay of a circuit to pass before capturing the result in the backup register, timing errors in that register can be prevented. Recovery takes only one clock cycle, during which the superscalar pipeline is allowed to continue execution of unaffected instructions. The area overhead for detection is kept low by re–using the combinational logic which makes up the pipeline stages, and only duplicating the registers between these stages. To further reduce the area penalty, only the stages with the longest delays need to be augmented with dual latches. Circuitry must also be added to perform the error recovery, but this is modest as well, since the logic involved is not complex and may re–use already existing signals in the pipeline. Overall, SPRIT^{3}E provides a viable means of tolerating timing errors.

CHAPTER 4. DYNAMIC FREQUENCY SCALING

To support the dual latching circuitry and maximize the performance of the pipeline, the main and phase shifted clock must be carefully generated. There are important considerations that determine the limits of the frequency scaling as well as the amount of the phase shift used to create the PS clock. The properties of the hardware used to generate the clocks must also be taken into account when determining the frequency modification scheme. Finally, the technique used in sampling the timing errors of the system also has implications for the achievable performance improvement.

4.1 Clock Timing Considerations

The timing error tolerance provided by the SPRIT^{3}E hardware requires support from precise clock generation. As the main clock frequency is allowed to scale higher and higher, the phase shift of the PS clock must be increased. The factors that influence the clock generation will be discussed by considering figure 4.1. The figure presents an example of different allowable main and PS clocks for a circuit. In this example, the worst case propagation delay of the circuit is 10 ns, which yields a clock frequency of 100 MHz. The contamination delay, which measures the earliest time at which the outputs may change following a change in the inputs, is 3 ns for this circuit. To begin operation, the periods of both the main and PS clocks are set to the worst case propagation delay of 10 ns. This is shown in the initial row at the top of the figure. At this speed, no timing errors will occur; the same value will be latched into both the main and backup registers. Observing no errors, the frequency is allowed to scale. As the frequency becomes faster, the PS clock is phase shifted. The mid row in the figure presents the situation after some scaling has taken place. The clock period has been shortened to 9 ns.

To compensate, the PS clock is phase shifted by 1 ns. By doing this, the circuit is effectively given from the first rising edge of the main clock at time 0 to the second rising edge of the PS clock at time 10 to execute, a full propagation delay. It may appear that because the rising edge of PS clock at time 10 occurs after a rising edge of the main clock, the data may be incorrect. However, although new inputs are given to the circuit at time 9, the data stored in the backup register will correspond to the old inputs provided in the previous cycle. This is ensured because of the contamination delay of the circuit. Although the inputs to the circuit have changed at 9 ns, a change in outputs won't be seen until 12 ns, well after the backup register captures at time 10.

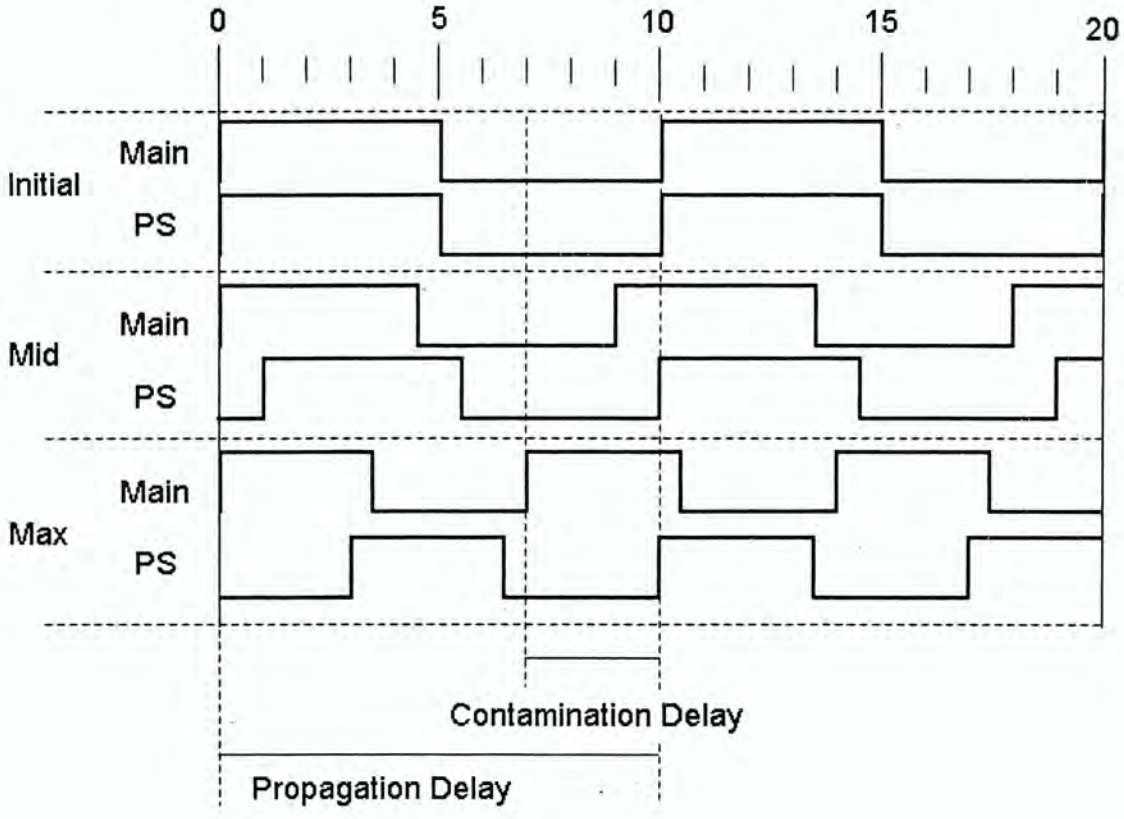

Figure 4.1 Examples of Main and PS Clocks

This important dependence on contamination delay leads directly to the limitation of the frequency scaling. The third row in figure 4.1 shows the maximum achievable frequency. At a period of 7 ns, the PS clock must be shifted by 3 ns. As shown, this phase shift lies at the

boundary of the contamination delay. Any further scaling of the frequency will cause the phase shift of the PS clock to be pushed past the contamination delay, and may introduce errors.

The period in this example is allowed to scale from 10 ns to 7 ns, giving a maximum improvement of 30%. In general, the maximum improvement, dependent on the propagation delay, t_{pd}, and the contamination delay, t_{cd}, is given by equation 4.1. Because the amount of exploitable performance depends directly on the contamination delay, pipeline stages with short contamination delays will see less possible improvement. If the contamination delay of a stage is limiting the performance of SPRIT^{3}E, that stage may be redesigned to increase its contamination delay without affecting its propagation delay. At first, this redesigning seems as simple as adding delay buffers to the shortest paths through the stage. However, since the delay of a stage is strongly input dependent, and because the shortest and longest paths often overlap, increasing the contamination delay without changing the worst case delay is a not a trivial issue. Still, although SPRIT^{3}E is bounded by contamination delay, it is the occurrence of timing errors that should determine the true amount of frequency scaling.

$$\text{Maximum Speedup} = \frac{t_{cd}}{t_{pd}} \tag{4.1}$$

4.2 Clock Generation

The dynamically tuned frequency is achieved through the global feedback system pictured in figure 4.2. Before operation begins, a small, non–zero, error rate is programmed as the set point. The clock controller is initialized with the worst case delay parameters of the pipeline. As stated above, the initial frequency of the clocks is the worst case propagation delay, and the PS clock begins with no phase shift. These values are sent to the clock generator block. This block consists of a voltage controlled oscillator (VCO) in series with 2 digital clock managers (DCMs). The VCO is able to generate a variable frequency clock to meet the value given by the clock controller. The first DCM locks the output of the VCO to provide the main clock to the pipeline. The second DCM provides a dynamically modifiable phase shift. It takes the main clock as well as the value requested by the clock controller and produces the PS clock.

Both DCMs provide a *locked* output as well, which is used to determine when the main and PS clocks have regained stability. During the period in which the clocks are being adjusted, the pipeline must be stalled. To avoid a high overhead from frequent clock switching, the number of timing errors in the pipeline will be sampled at a large interval and a new frequency determined. A detailed discussion of different sampling methods is given below in 4.3. These methods amortize the penalty incurred by switching clock frequencies over a long execution period.

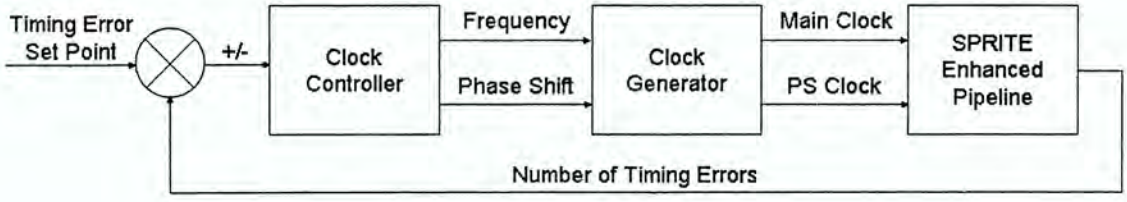

Figure 4.2 Feedback Control System Used to Tune Clock Frequency

4.3 Error Sampling Methods

An important design consideration of the feedback control system is the method which is used to sample the timing errors occurring in the system. In order to measure the error rate, a history of the errors that have occurred in past cycles must be kept. The different methods for error sampling vary in how this history is maintained and the frequency at which it may be checked. The methods can be generally thought of as discrete, continuous, or semi-continuous. The size of the window that determines the length of the error history affects all three methods. In the following discussion, a window of 100,000 processor cycles is used, as this length allows for an accurate estimation of the error rate. Also, the amount by which the period is allowed to scale each time is a consideration.

In the discrete method, the error history is checked once per window, and gives a count of the errors which have occurred since the last window. This value is kept by a single counter that increments every cycle in which an error occurs. When the window of 100,000 cycles passes, the counter is checked, and depending on whether the error rate is greater or less than

the set point, the clock period may be incremented or decremented. The error counter is then cleared, and begins to count the errors occurring in the next window of 100,000 cycles. The maximum size needed for the counter in the discrete case is 17 bits. Compared with the other sampling methods, this is a relatively small overhead. However, the performance of this method suffers because the period can only be updated once in every error window. To lower the number of counter bits needed, as well as prevent performance degradation from a large amount of errors occurring before the window expires, it is possible to set a limit on the number of errors that will be tolerated. For instance, using an upper limit of 5%, no more than 5,000 errors can occur in a window of 100,000 cycles. If this value is ever reached, the period will immediately be raised and both the counter and error window reset, regardless of the current window. Through setting this limit, the counter can be implemented with only 13 bits.

On the other side of the spectrum, the continuous method uses a sliding window of 100,000 cycles to maintain the history of errors. To implement this window, a 100,000 bit shift register is used, one bit for every cycle in the window. Each cycle, a bit is shifted into the register corresponding to the occurrence of an error in that cycle. The current number of errors present in the sliding window are maintained in a counter. This can be easily accomplished by comparing the value shifted out from the shift register with the value being shifted in. If these values are the same, the counter need not be changed. If a 1 is shifted in, and a 0 is shifted out, then the total number of errors has increased. Alternately, when a 0 is shifted in and with a 1 shifted out, the total number of errors in the window should be decreased. Because the count maintains a continuous history, the errors may be checked at any point and a valid error rate can be used to determine if the clock period should be changed. However, changing the period is a lengthy operation, and as such should not be performed too often. As a result, the minimum time between period switching should be limited. This method allows a very short time between period switches, while at the same time maintaining a long history. This means that the clock period is allowed to scale quickly to the optimal value, and will stabilize there. However, continuous error sampling achieves this performance improvement at an impractically high area cost. There are 100,000 bits needed for the shift register as well 17

bits for the counter. The later may be improved as in the discrete case by setting an upper bound on the amount of tolerable errors. However, it is less likely that this upper bound will be reached in the relatively few intermediate cycles compared with the 100,000 cycles that separate switches in the discrete case.

In order to obtain similar benefits to the continuous case, yet avoid its high overhead, a third, semi–continuous method may be used. In this method, the error window is divided in to 5 counters. Each counter maintains the total errors occurring in a separate 20,000 cycles of the error history. The counters are used in a rotating fashion, so that at every sampling, the oldest counter is cleared and begins counting. In this way, the errors may be checked every 20,000 cycles and the error rate determined using a history of 100,000 cycles. Thus the switching time is much closer to the minimum allowed by the continuous method. At the same time the overhead, while larger than the discrete case, is reasonable. Each counter needs 15 bits, so for the 5 counters, 75 bits will be required. As in both previous cases, if an upper bound is set on the number of errors that may occur, then fewer bits will be needed for the counters. Should the upper bound be reached, the period will be immediately increased, and the oldest counter will be cleared and begin to record new errors.

When considering any error window, there is a trade off between the frequency allowed for sampling and the number of bits needed to store the history of errors in the window. Table 4.1 presents a summary of the each of the error sampling methods. In chapter 5, all three error sampling techniques will be evaluated, though the continuous technique should only be viewed in a theoretical light as an upper bound on the achievable performance.

Table 4.1 Summary of Error Sampling Methods

Method	Window	Minimum Switching Time	Bits to Store Error History
Discrete	100,000	100,000	17
Continuous	100,000	None	100,017
Semi–Continuous	100,000	20,000	75

Finally, the amount by which the period is changed at every switching time is also a design consideration. A basic and relatively simple to implement technique scales the period by a

constant amount. This method requires storing an upper and a lower threshold for errors. If the errors occurring in a window are less than the lower threshold, the period is decreased, causing more errors to occur. If the period is lowered too far, then the number of errors may become greater than the upper threshold, and the period must be increased. In this way, the steady state, optimal period will be attained. In order to allow the period to stabilize within the thresholds, however, the amount by which the period is changed must be small to avoid frequent flucuations when nearing the optimal point. This leads to decreased performance from initial cycles in which the period is far from optimal and is changing slowly. To address this problem, an error sampling technique may implement more than two thresholds. Using multiple thresholds, error rates falling further from optimal will allow greater changes in the period. As the period approaches the optimal, smaller changes occur to fine tune the operating frequency. This method fast forwards the period through initial cycles and can improve performance, but does require more complex hardware to implement the period switching.

4.4 Analysis

In theory, the advanced clocking system used to provide the main and PS clocks will allow the SPRIT^{3}E pipeline to operate free of faults even as timing errors occur. There are a couple issues in practice, however, that must be taken into consideration. First, as it has been observed in this thesis that the worst case propagation delay rarely occurs in implementation, it follows that the contamination delay in the implemented circuit may be shorter than estimated. Because the length of the contamination delay is vital to the correctness of the backup register, it should be estimated using *best case* parameters. That is to say, a variation in the observed contamination delay can only make it longer than the estimated value. If this estimate gives a contamination delay that limits performance of the clock scaling, it can be increased using buffer insertion as described above. The second issue is that of clock skew. The effects that lead to variable circuit delays, such as temperature, voltage, and process variations, also cause variations in the clock period, referred to as clock skew. In SPRIT^{3}E, skew in the main clock is not an issue, since timing errors are expected to occur in the main register. However, it is

important that skew in the PS clock does not cause a delay from main to PS clock edges that is longer than the contamination delay. In order to account for this possibility, the worst case clock skew should be assumed when determining the maximum frequency scaling achievable. Having accounted for these considerations, the clocking system presented in this chapter can reliably support the dual latching framework and give a pipeline the ability to scale faster than its worst case performance.

CHAPTER 5. EXPERIMENTAL RESULTS

To gauge the performance improvements provided by the SPRIT^{3}E framework, a sequence of experiments has been performed. An initial study of a simple multiplier circuit established that significant room for improvement does indeed exist. From there, applications executing on a superscalar processor were analyzed, and the effects of augmenting the pipeline with SPRIT^{3}E were calculated.

5.1 Multiplier Circuit Analysis

As a first step in evaluating this technique, the frequency induced timing errors of a multiplier circuit were observed. The circuit was implemented in a Xilinx XC2VP30 FPGA, Xilinx (2005). A block diagram of the system is shown in figure 5.1. As presented in previous chapters, the main and PS clocks operate at the same frequency, with a phase shift between them. However, in this circuit, the period of the clocks remains constant at the worst case delay. The phase shift of the PS clock latches the multiplier result in the early register after a shorter delay. In operation, every rising edge of the main clock triggers two linear feedback shift registers to provide random inputs to the multiplier logic. To minimize the routing delays, an 18x18 multiplier block embedded into the logic of the FPGA was used. The output will be latched first by the early register, and a phase shift later by the main register. Error checking occurs at every cycle, and is pipelined to allow maximum shifting of the PS clock. A finite state machine (FSM) is used to to enable the error counter for 10,000 cycles. To prevent the counter from counting errors that may occur when initializing the pipeline, the FSM begins enabling after 4 delay cycles have passed.

The worst case propagation delay of the synthesized circuit was estimated at 6.717ns by

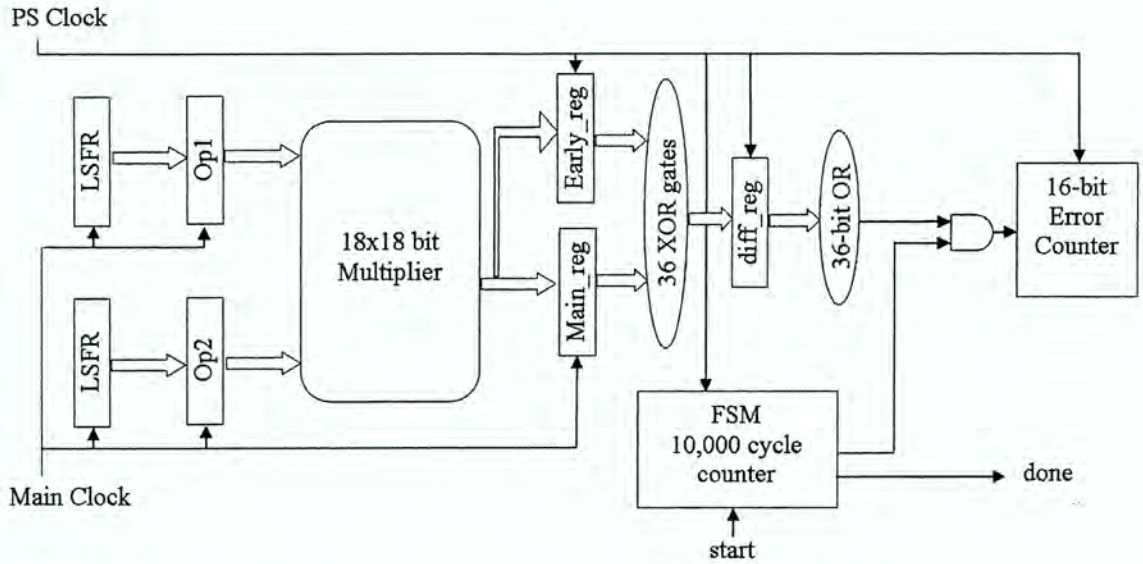

Figure 5.1 Block Diagram of the Multiplier Circuit

the timing analyzer. To allow plenty of time for the circuit to execute before being captured in the main register, a clock period of 8ns was used. The phase shift of the PS clock was varied from 0 to -5.5 ns, giving effective clock periods of 8 to 2.5ns. For each effective period, the total errors were counted for an execution run of 10,000 cycles. By counting the number of cycles that produce an error for a given phase shift of the PS clock, the rate of timing errors that will occur when operating at that effective period can be calculated. For example, when the PS clock is shifted such that its rising edge occurs 5ns before the main clock, the multiplier logic is effectively being given 3ns to compute. At this frequency, about 94% of the 10,000 cycles produced a timing error.

Figure 5.2 presents the percentage of cycles that produce an error for different effective clock periods. As shown, although the worst case delay was estimated at 6.717ns, the first timing errors don't begin occurring until a period of under 4ns. In fact, running at 3.75ns will produce an error in only 1% of the total cycles. This shows the effect of the margins that have been added to account for worst case operation. Using a method such as dual latching to tolerate a small amount of timing errors would allow this circuit to run at almost half the period, a speedup of 44%.

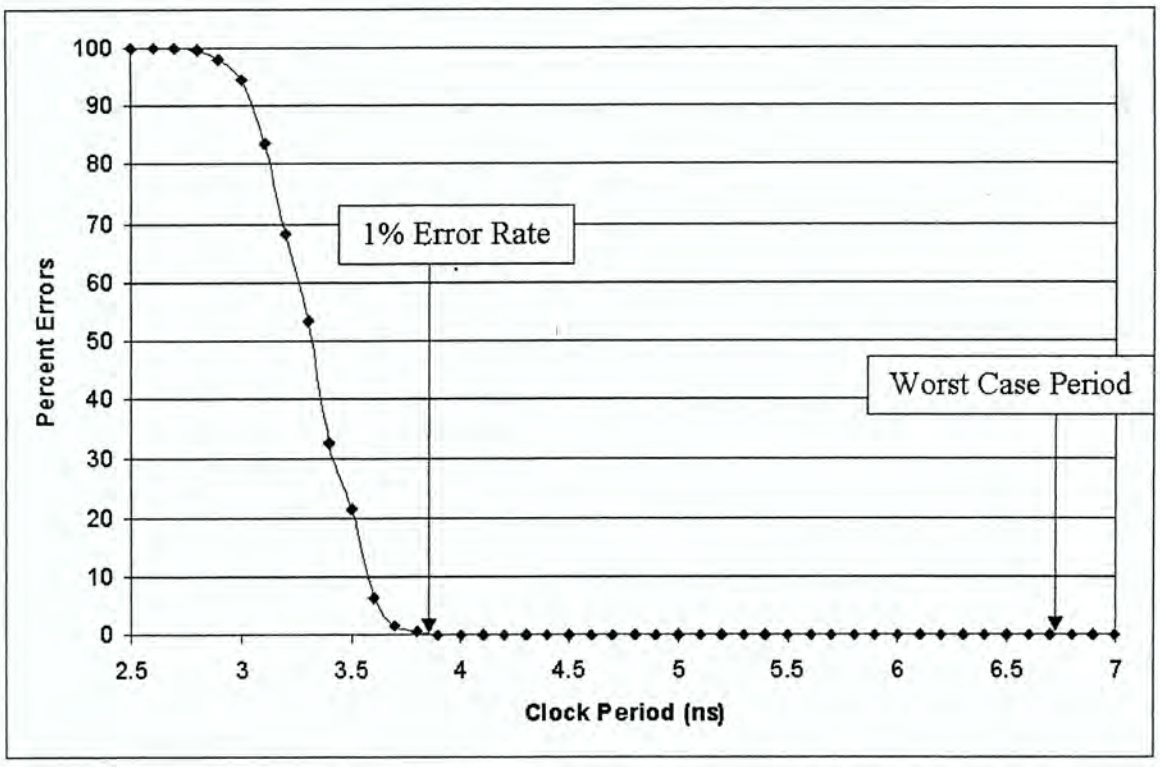

Figure 5.2 Percent of Error Cycles Versus the Clock Period for the Multiplier Circuit

5.2 Superscalar Processor Analysis

Having observed the large effect of worst case margins in the multiplier circuit, the next step was to evaluate the ability of the SPRIT^{3}E framework to exploit these margins and improve the performance of a superscalar processor. First, a suitable processor was chosen and implemented in the Virtex II Pro FPGA. Then, the frequency induced timing error behavior of the processor was analyzed using a method similar to the multiplier circuit. Finally, using the error rates obtained from execution in the FPGA, the effects of using the SPRIT^{3}E framework were evaluated for different error sampling methods.

For this experiment, a modified verion of the DLX superscalar processor, available from Horch (1997), was used. The relevant parameters of the processor are summarized in table 5.1. The DLX features a 2 instruction wide issue, decode and commit stage. A 5 entry reorder buffer allows program instructions to execute out of order on 4 funtional units: an arithematic

logic unit (ALU), a multiply divide unit (MDU), a branch resolve unit (BRU), and a load store unit (LSU). Branch prediction is provided in the form of a 4 entry branch target buffer. The instruction and data caches each provide 64 bytes of storage and may be accessed in 1 cycle. An additional 64 kilobytes of instruction and 64 kilobytes of data storage are available, which takes 2 cycles to access.

Table 5.1 Relevant Parameters of the DLX processor

Parameters		**Value**
Decode / Issue / Commit bandwidth		2
Reorder Buffer Entries		5
Number of Function Units	ALU	1
	MDU	1
	BRU	1
	LSU	1
Instruction and Data Cache Size(Bytes)		64
Memory Size (KBytes)		64

The superscalar DLX processor was synthesized for the Xilinx XC2VP30 FPGA. The maximum timing delay between registers in this circuit was 21.982ns, between the source registers of the MDU and the data registers of the reorder buffer. Similar delays, all around 20ns, exist through the other function units to the reorder buffer, as well as from the dispatch stage to the reorder buffer. Thus, to analyze the timing error rates of the processor, the reorder buffer registers were augmented with additional registers as well as the comparing and counting circuitry shown in figure 5.1. The processor was operated at varying phase shifts of the PS clock, and the percentage of cycles in which an error occured for the execution run was recorded.

Figure 5.3 shows the error rates of operating the DLX at effective periods between 15 and 3.5ns for 3 different benchmarks. The RandGen application performs a simple random number generation to give a number between 0 and 255. One million random numbers are generated, and the distribution of the random variable is kept in memory. The MatrixMult application multiplies two 50x50 integer matricies and stores the result into memory. The BubbleSort program performs a bubblesort on 5,000 half–word variables. For this application the input is given in the worst case unsorted order. As shown in the figure, for both RandGen and

MatrixMult, the errors become significant at around 8.5ns, while the error rate of BubbleSort stays low until around 8ns. This is because both the MatrixMult and RandGen applications use the MDU, and thus are likely to incur the worst case path. The BubbleSort uses only the ALU to perform comparisons as well as addition and subtraction, so it is able to operate at lower periods before errors begin to occur.

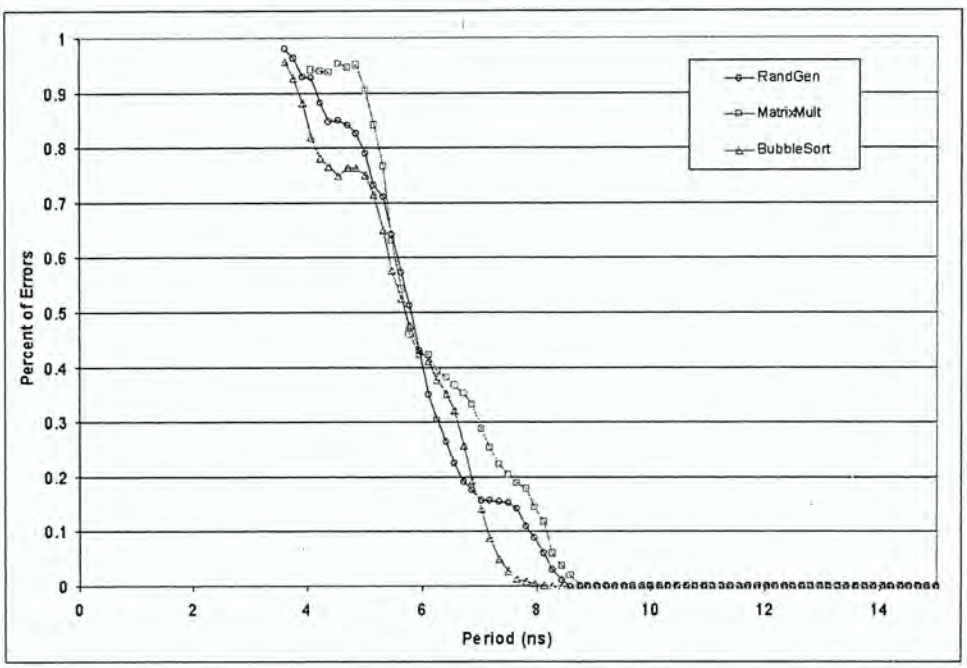

Figure 5.3 Percent of Error Cycles Versus the Clock Period for the DLX Processor

Using the probability distribution for the error rate determined in the previous step, a simulator was written to evaluate the effectiveness of the SPRIT^{3}E framework using the different methods of error sampling discussed in chapter 4. The simulator runs on a cycle by cycle basis. At every cycle, the probably of a timing error is determined based on the current period, which in turn is used to decide if a timing error occurs that cycle. The timing errors are stored differently depending on which error sampling technique is being used. The clock period is allowed to change after a certain number of cycles have passed, which is also dependent on the sampling method. In this experiment, the amount by which the clock period is allowed to change is held constant. Each benchmark is evaluated separately, and may be executed for

either its original number of cycles, reported in table 5.2, or for a long run of 120 million cycles.

Table 5.2 Length of Applications in Cycles

Application	Cycles to Execute
MatrixMult	2901432
BubbleSort	118896117
RandGen	15750067

The scaling behaviour for the matrix multiplier application executed for a long run is shown in figure 5.4. The figure highlights the differences between discrete, semi–continuous, and continuous sampling. The other applications show similar period scaling over the course of execution. As the figure demonstrates, the long intervals between switching for the discrete sampling method prevent it from reaching the optimal period as quickly as the continuous and semi-continuous cases. This has important implications for the performance gains achievable by the discrete sampling method.

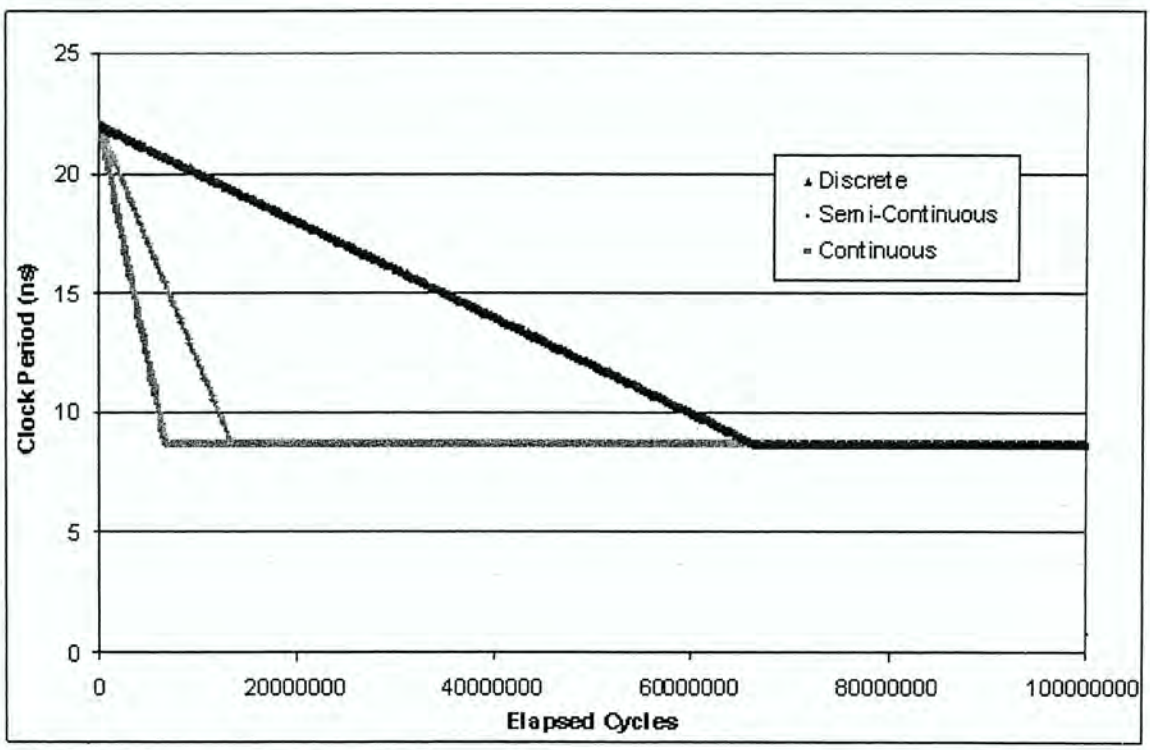

Figure 5.4 Dynamically Scaled Clock Period Versus the Elapsed Cycles for MatrixMult

As the simulation is running, the execution time of the application is calculated. The reference execution run sets the period at the worst case value and allows no scaling, thus no timing errors occur. For the other cases, each cycle in which a timing error occurs results in a stall cycle being injected into the pipeline. Also, when a change in period occurs, the time taken to lock the DCMs to the new frequency is added to the total execution time. The execution times for each application when run for its original execution cyles is shown in figure 5.5, normalized to the reference, worst case time. The BubbleSort application shows the best performance, as it runs the longest and thus runs the longest at the optimal period for any sampling method. The MatrixMult application, however, is only long enough for gains achieved by lowering the period to begin to outweigh the penalties for doing so. Each benchmark was also evaluted running for a long execution time. The performance results are presented in figure 5.6. For this variation, all benchmarks perform similarly, with the discrete error sampling method giving on average a 57% improvement over the worst case, and the semi–continuous and continuous methods outperforming it at 44% and 43% respectively.

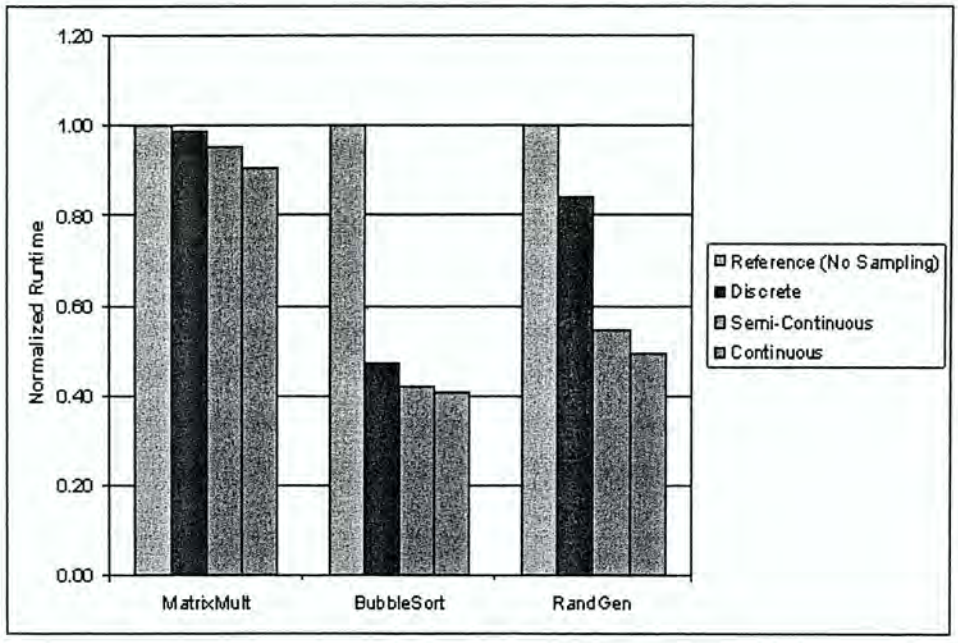

Figure 5.5 Relative Performance Gains for Different Applications Executing For Original Cycles

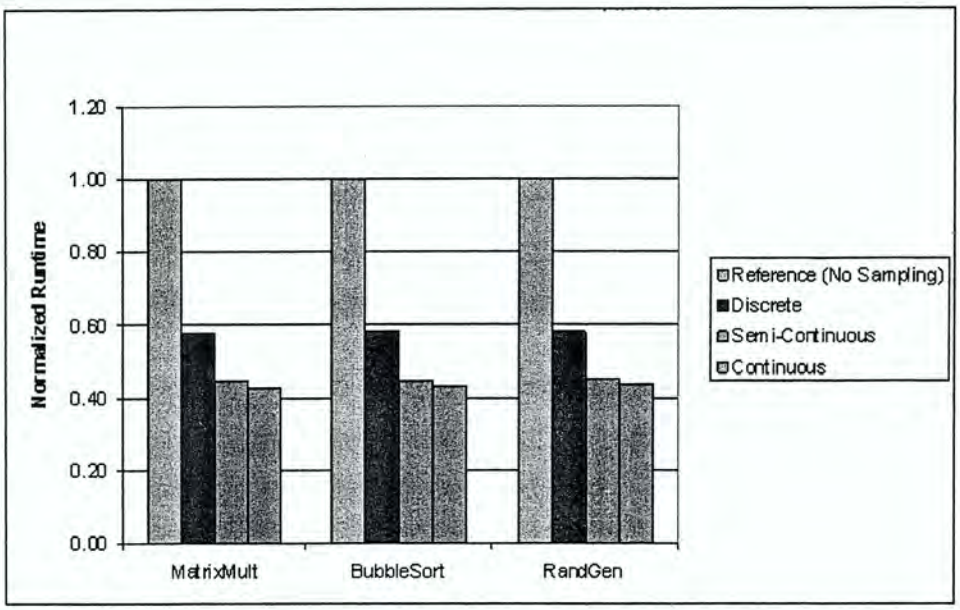

Figure 5.6 Relative Performance Gains for Different Applications Executing Over the Long Run

CHAPTER 6. CONCLUSIONS

As demonstrated by successful overclocking, the current practice of assuming the worst case delay to set the frequency for synchronous circuits is far too conservative. At the same time, fault tolerance is necessary to ensure reliability if timing errors are not avoided with worst case margins. The SPRIT^{3}E framework addresses these problems with relatively simple additions to the superscalar pipeline. Only the pipeline registers are duplicated, and the large combinational logic blocks making up the stages are reused by utilizing temporal redundancy. Addtional overhead comes from the error recovery logic, but this too may be kept modest by reusing existing pipeline signals whenever possible. All in all, the performance gained by operating at the optimal, sub–worst case period more than justifies the overhead of the detection and recovery logic.

Although the experimental results measured the timing errors generated in an FPGA, an extension can be made to logic implemented in ASIC technology. In both cases, the variables that govern the delay as well as the likelihood of inputs sensitizing the longest delay path are the same. Thus, even in the ASIC case, the selected clock period will be overly conservative. However, unlike in ASICs, circuits implemented in FPGAs must use general routing resources to implement wires. As a result, both the estimated and actual delays of FPGA circuits will be longer than for ASIC circuits in similar technology. So, although the traditional worst case desgin methodology will cause both methods of implementation to suffer from an overly slow period, the actual timing error trends determining the amount of exploitable performance improvement will differ.

This work presents an initial exploration of the possiblities for taking advantage of the margins produced by worst case design mentality. In the future, implementing a main memory

system for the synthesized DLX processor would allow full scale benchmarks to be evaluated, as well as allow an exploration of the effect of increasing the clock frequency on the average instructions committed per clock cycle. Also, as the propagation delay is dependent on the operating temperature, an interesting study could gauge the effectiveness of different error sampling methods in a variable temperature environment. Finally, further extensions to toleration of particle induced SEUs could be explored using fault injection, and may warrent an augmentation to the SPRIT^{3}E framework. This thesis presents a very promising technique, with many exciting directions for the future.

Bibliography

Allan, A., Edenfeld, D., Joyner, Jr., W. H., Kahng, A. B., Rodgers, M., and Zorian, Y. (2002). 2001 technology roadmap for semiconductors. *J-COMPUTER*, 35(1):42–53.

Amde, M., Blunno, I., and Sotiriou, C. P. (2003). Automating the design of an asynchronous dlx microprocessor. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 502–507, New York, NY, USA. ACM Press.

Austin, T., Bertacco, V., Blaauw, D., and Mudge, T. (2005). Opportunities and challenges for better than worst-case design. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 2–7, New York, NY, USA. ACM Press.

Colwell, B. (2004). The zen of overclocking. *IEEE Compututer*, 37(3):9–12.

Das, S., Pant, S., Roberts, D., Lee, S., Blaauw, D., Austin, T., Mudge, T., and Flautner, K. (2005). A self-tuning dvs processor using delay-error detection and correction. In *IEEE Symposium on VLSI Circuits*.

Emerson, K. (1997). Asynchronous design—an interesting alternative. In *IVLSI:97*, pages 318–320.

Ernst, D., Kim, N. S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K., and Mudge, T. (2003). Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 7, Washington, DC, USA. IEEE Computer Society.

Frank, D. J., Dennard, R. H., Nowak, E., Solomon, P. M., Taur, Y., and Wong, H.-S. P. (2001).

Device scaling limits of si mosfets and their application dependencies. In *Proceedings of IEEE*, volume 89, pages 259–288, Washington, DC, USA. IEEE Computer Society.

Hartstein, A. and Puzak, T. R. (2002). The optimum pipeline depth for a microprocessor. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 7–13, Washington, DC, USA. IEEE Computer Society.

Hauck, S. (1993). Asynchronous design methodologies: An overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle.

Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. (2001). The microarchitecture of the pentium 4 processor. Technical report, Intel.

Horch, J. (1997). A superscalar version of the dlx processor. http://www.rs.e-technik.tu-darmstadt.de/tud/res/dlxdocu/superscalardlx.html. date accessed: February 10, 2006.

Kim, S. and Somani, A. K. (2001). Ssd: An affordable fault tolerant architecture for superscalar processors. In *PRDC*, pages 27–34.

Liu, T. and Lu, S.-L. (2000). Performance improvement with circuit-level speculation. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 348–355, New York, NY, USA. ACM Press.

Mavis, D. G. and Eaton, P. H. (2002). Soft error rate mitigation techniques for modern microcircuits. In *International Reliability Physics Symposium*, pages 216–225.

Mirapuri, S., Woodacre, M., and Vasseghi, N. (1992). The mips r4000 processor. *IEEE Micro*, 12(2):10–22.

Nassif, S. R. (2001). Modeling and forecasting of manufacturing variations (embedded tutorial). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 145–150, New York, NY, USA. ACM Press.

Nickel, J. B. and Somani, A. K. (2001). Reese: A method of soft error detection in microprocessors. In *DSN '01: The International Conference on Dependable Systems and Networks*, pages 401–410.

Overclockers.com (2006). Overclockers website forum. http://www.overclockers.com. date accessed: April 10, 2006.

Shivakumar, P., Kistler, M., Keckler, S. W., Berger, D., and Alvisi, L. (2002). Modeling the effect of technology trends on the soft error rate of combintational logic. In *DSN '02: The International Conference on Dependable Systems and Networks*, pages 389–398.

Uht, A. K. (2000). Achieving typical delays in synchronous systems via timing error toleration. Technical Report 032000-0100, Department of Electrical and Computer Engineering, University of Rhode Island.

Uht, A. K. (2005). Uniprocessor performance enhancement through adaptive clock frequency control. *IEEE Trans. Comput.*, 54(2):132–140.

Von Neumann, J. (1966). *Automata Studies.* Number 34. Princeton University Press, Princeton, New Jersey.

Weaver, C. and Austin, T. (2001). A fault tolerant approach to microprocessor design. In *DSN '01: The International Conference on Dependable Systems and Networks*, pages 411–420.

Weaver, C., Gebara, F., Austin, T., and Brown, R. (2002). Remora: A dynamic self–tuning processor. Technical Report CSE-TR-460-02, University of Michigan.

Xilinx (2005). Virtex-ii pro and virtex-ii pro x fpga user guide. http://www.xilinx.com. date accessed: April 17, 2006.

Yeap, G. C.-F. (2002). Leakage current in low standby power and high performance devices: trends and challenges. In *ISPD '02: Proceedings of the 2002 international symposium on Physical design*, pages 22–27, New York, NY, USA. ACM Press.

ACKNOWLEDGEMENTS

The research reported in this paper is partially supported by NSF grant number 0311061 and the Jerry R. Junkins Endowment at Iowa State University.